A Tale of Two Case Studies: A Unified Exploration of Rust Verification with SeaBMC

Joseph Tafese*
University of Waterloo
Waterloo, Canada
jetafese@uwaterloo.ca

Siddharth Priya*
University of Waterloo
Waterloo, Canada
siddharth.priya@uwaterloo.ca

Giuliano Losa
Stellar Development Foundation
San Fransisco, USA
giuliano@stellar.org

Arie Gurfinkel
University of Waterloo
Waterloo, Canada
agurfink@uwaterloo.ca

Graydon Hoare
Stellar Development Foundation
San Fransisco, USA
graydon@stellar.org

***Abstract*—The Rust type system provides strong compile-time guarantees. However, some properties cannot be fully verified by the compiler. Specifically, properties like panic freedom and memory safety in mixed safe-unsafe code require verification beyond what the language enforces. We explore how to verify these properties in real-world Rust code using SeaBMC, a bounded model checker that ingests LLVM-IR generated by the Rust compiler. We demonstrate our approach through two case studies. In the first, we develop unit proofs for functional properties of four data-structure libraries: smallvec, tinyvec, seavec, and result-type from the Rust standard library. These unit proofs are checkable by both SeaBMC and Kani, a state-of-the-art bounded model checker for Rust, and we find that SeaBMC verifies these units an order of magnitude faster than Kani. The second case study focuses on verifying panic freedom of Wasmtime's Winch compiler. This application is driven by the requirement for high reliability when compiling WASM smart-contracts in the Stellar network. This case study highlights that executable counterexamples from SeaBMC are highly effective for localizing issues and discovering invariants. Our main contributions are (1) a new tool for Rust verification which, on our benchmarks, is an order of magnitude faster than Kani, (2) two case studies with reusable benchmarking and testing infrastructure, and (3) practical guidelines stemming from our experience verifying real-world code-bases.**

I. Introduction

The Rust type system makes progress on long standing safety problems in systems programming—most notably, it ensures the absence of memory-safety violations, data races, and other undefined behavior. However, the Rust type system guarantees those safety properties only for a subset of the language called Safe Rust. For expressiveness and speed, programmers must often resort to Unsafe Rust. Thus, the Rust type checker does not guarantee the safety of many practical Rust programs.

Verification tools that work on mixed safe-unsafe Rust code can fill an important gap left by the Rust type system. In practice, undefined behavior stemming from unsafe Rust code may lead to panics (an exception denoting an unrecoverable error), crashes, or subtle memory-corruption issues. We treat

```
fn main() {                                                          1
  let arr = [10, 20, 30, 40, 50];                                    2
  let p = arr.as_ptr();                                              3
  // Move pointer FAR past the array bounds                          4
  let bad_ptr = unsafe { p.add(10) };                                5
  let bad_index = unsafe { bad_ptr.offset_from(p) } as usize;        6
  println!("Value: {}", arr[bad_index]);                             7
}                                                                    8
```

Fig. 1: Verifying panic freedom in the presence of unsafe code requires low-level reasoning.

panics and undefined behavior interchangeably, unless specifically mentioned. For example, in Fig. 1, an unsafe Rust block creates an invalid pointer at line 5, which leads to a runtime panic at line 7. This style of programming adds security defensively. However, a program may wish to ensure no panics occur, i.e., panic freedom. Thus, panic freedom becomes an important property to prove.

In this paper, we present a verification methodology for mixed safe-unsafe Rust code based on the SeaBMC bounded-model checker. SeaBMC is part of the SeaHorn verification toolkit. It analyzes LLVM bitcode produced by the Rust compiler and massaged by SeaHorn's pre-processing passes, and relies on an SMT solver to discharge verification conditions. Since SeaBMC works on the low-level representation of the program post Rust compilation to LLVM, it can verify mixed safe-unsafe Rust, and even mixed Rust and C programs.

A SeaBMC user must assemble three components for verification. The first is the system under test (SUT). This may be a single function or a more complex arrangement of system components. The second is the runtime environment needed by the SUT. This may be part of the language's runtime system (memory allocation, error handlers, etc.), which we are not interested in verifying, but it may also contain system components that the SUT uses but that we are not verifying. During verification, the runtime environment is replaced by a simpler verification-only environment. Third, a unit proof [1] (similar to a unit test) that sets up concrete and symbolic inputs and state needed for the SUT to execute. It usually has three stages—setup pre-conditions, call SUT, check post-

* Authors made an equal contribution.

https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_33

conditions. We apply this methodology in two case studies.

In the first case study, we write 83 unit proofs that verify low-level functional correctness. We compare SEAHORN and KANI [2] on these unit-proofs. KANI is a bounded model checker for mixed safe-unsafe Rust code developed by AWS and based on CBMC [3], and it has recently seen a flurry of activity [4]. Both tools are configured to automatically check spatial memory safety. We show that SEABMC is faster than KANI by a factor of 10 or more under similar checks. Additionally, we provide a new design point around panics in unit proofs. KANI adopts the view that any panic is always an error (PANIC-ERROR). However, this can force users to write unit proofs with very strong pre-conditions in order to avoid *any* panic. An alternative is to abort execution upon a panic, without considering it an error (PANIC-ABORT). The PANIC-ABORT approach allows writing weaker pre-conditions quickly. The executions that panic simply exit early. This style of unit proofs is inspired by smart contracts written in Rust that use panics to safely terminate an incorrect execution early. In summary, SEABMC can meet the challenge of verifying low-level safe-unsafe Rust code by being precise and fast.

In the second case study, we deploy our methodology to verify parts of WINCH, an industrial WASM-to-executable compiler for various architectures. With the aim of identifying invariants that ensure panic freedom, we contribute 14 unit proofs that check properties of a system under test (SUT) consisting of roughly $3,300$ lines of Rust code (LOC). We use SEABMC to verify panic freedom of core compiler components since the usage context is especially sensitive to nondeterministic panics. During this exercise, we discover a panic that encodes a requirement: WINCH must run on an architecture with 64-bit pointers. Without assuming this pre-condition, SEABMC reaches a panic, and SEAHORN provides an executable counter example that makes it easy to discover the 64-bit pointer-size requirement.

We show that SEABMC meets the challenge of verifying Rust programs which mix both safe and unsafe code. The SEABMC tool and case studies are open source. The rest of the paper is organised as follows. In Section II, we describe how SEABMC works on LLVM bitcode. Section III describes how unit proofs for Rust are setup. In Section IV we describe how SEABMC compares to the state-of-art KANI tool. We provide our experience verifying parts of the WINCH compiler using SEABMC in Section V. Future and related work is discussed in Sections VI and VII, respectively. We conclude in Section VIII. Artifacts for both the VERIFY-RUST and WINCH case studies are available online[1].

II. ENGINEERING SEABMC FOR RUST

SEAHORN is a verification toolkit for LLVM programs. SEABMC is a bit-precise bounded model checking engine for SEAHORN that has been used in various studies ([1], [5], [6]). LLVM IR produced by a language frontend is transformed by the SEAHORN pre-processing pipeline into a representation where memory dependencies between LLVM operations are made explicit. This intermediate representation is called SEA-IR [7] and is the input for SEABMC. SEABMC was originally developed for verifying LLVM IR produced by the CLANG compiler for C. It works by symbolically unrolling a program's control flow-loops, function calls, and conditional branches, up to a user-specified bound. It then generates verification conditions that capture all reachable states within that bound. These conditions are translated into SMT formulas and passed to a solver (Z3 [8] by default) which checks whether they are satisfiable, which would indicate that the program can reach an error state, e.g., a panic or an invalid memory access. When SEAHORN determines that an error state is reachable, it generates a harness that drives the SUT to the error state; after compiling and linking this harness with the SUT, the user can diagnose the issue with a standard debugger. For example, if a 32-bit symbolic value must be zero to trigger the panic, SEAHORN generates a harness that provides a 32-bit zero value. We will discuss this with an illustrative example in Section V-B.

For memory safety, SEABMC uses fat-pointers to carry allocation start and end addresses (bound) apart from pointed-to address. We also use shadow-memory to encode invariants such as allocation bounds, memory read/write, and no use-after-free. Moreover, the SEABMC memory allocator guarantees that fresh memory allocations are disjoint in the address-space from existing allocations. With this, pointer provenance reduces to checking if a fat pointer is within its allocation bound. More details can be found in [7].

Using SeaBMC with Rust typically involves compiling the code to a supported LLVM version (e.g., Rust v1.64 for LLVM 14). No modification of SeaBMC itself is required, since the tool treats LLVM IR uniformly regardless of whether it originated from C or Rust. Note, our experience is different from [9] where some previously unhandled LLVM instructions had to be handled for Rust. In our case, all relevant LLVM instructions generated by the Rust compiler are handled by SEABMC out-of-box. This compatibility illustrates a broader point: verification tools built for C can often be applied to Rust with little effort, provided they operate at the LLVM-IR level. However, some engineering work is needed to massage Rust compiler LLVM output to the right form for verification.

First, we have to simplify the Rust startup code. This code is responsible for setting up the runtime environment, and it executes before the Rust unit proof is called. The startup logic can be complex to execute symbolically. Moreover, the complex startup code is not needed for verifying the unit proof since SEABMC provides simpler stub implementations for this functionality. To remove this startup code, one option is to remove the target functions from the LLVM bitcode generated by the Rust compiler. This method used in [10] has limited success since a list of candidate functions to remove has to be maintained across Rust versions. Instead, we use a novel mechanism that treats the rust code as a library and links it to a C function that serves as the entry point for verification. During linking, Rust does not embed the unneeded functions

[1]https://doi.org/10.5281/zenodo.16415667

```
#[cfg_attr(kani, kani::proof)]
#[cfg_attr(kani, kani::unwind(5))]
#[cfg_attr(kani, kani::should_panic)]
#[no_mangle]
fn test_push() {
  const CAP : usize = 4;
  let mut v: ArrayVec<[u32; CAP]> = ArrayVec::new();
  let len: usize = verifier::any!();
  verifier::assume!(len <= CAP);

  for i in 0..len {
    v.push(verifier::any!());
    verifier::vassert!(v.len() == i + 1);
  }

  verifier::vassert!(v.capacity() == CAP);

  if len == CAP {
    // Vector is at capacity, so push should panic.
    v.push(verifier::any!());

    // This assertion should not be reachable
    // since the previous push panics.
    verifier::error!();}
}
```

Fig. 2: Unit proof for the TINYVEC `push` operation.

since the C runtime is designated active. This works well because SEAHORN already provides a runtime environment for a C executable [2].

Second, getting access to a LLVM bitcode representation of the complete Rust program is not straightforward. By default, the Rust compiler does not provide information on all the intermediate LLVM bitcode files from different compile units before linking. For example, the `emit-llvmir` option does not provide definitions of the allocation functions from the standard library. Without these definitions, the obtained LLVM code is incomplete and would require SEAHORN to patch the LLVM bitcode with missing definitions. This patching is not a clean or easily maintainable solution. Thus, we must obtain a complete LLVM program. For this, we rely on LLVM's FatLTO [11] feature. FatLTO produces a native binary that additionally embeds a complete LLVM program, which we then extract and input to SEAHORN.

III. VERIFYING RUST

We want SEABMC to be a testbed for exploring new ideas in verification of Rust programs. Towards this goal, we have created a benchmarking suite of Rust unit proofs with three goals in mind – portability, comparison to state-of-the-art, and, extensible experiments.

Unit proofs and CAS. Recall that a unit proof sets up concrete and symbolic inputs, pre-conditions, calls the system under verification (SUT), and checks the post-conditions. To make writing pre-and-post conditions natural to developers, it is useful to use code as specification (abbrev. CAS) [12]. We want unit proofs in VERIFY-RUST to be tool agnostic. For this, we have developed VERIFIER-LIB, a Rust crate that provides a single API to write SEABMC and KANI specifications as assumptions and assertions in the unit proof directly.

An example unit proof for the TINYVEC `push` operation, shown in Fig. 2, illustrates these concepts. Lines 1–4, are configuration options for KANI. The first line marks the function as a KANI proof. The second line sets the loop unwinding bound to `5`. The third line marks that the code is expected to panic. KANI treats all panics as verification failures otherwise. The `#[no_mangle]` attribute tells the Rust compiler to not mangle [13] the function name. It is required because SEAHORN verification infrastructure uses the LLVM function as a key to run and display test results. Going to the unit proof itself, lines 6–9 setup the unit proof preconditions. The vector capacity is set to four elements. The length is designated as a non deterministic value using `verifier::any!()`. It is then constrained to between zero and capacity inclusive. Note that the `verifier::assume!` and `verifier::vassert!` macros are from VERIFIER-LIB. The specification is delegated to corresponding SEABMC or KANI API when the unit proof is compiled. Lines 11–14 call the SUT in a loop. After each call, the unit proof checks that the vector length has indeed increased by one. Finally line 16 checks that the capacity is unchanged. Now, an additionally property of TINYVEC `push` is that is should panic if the vector is already at capacity. This is checked in lines 18–24. Here, the unit proof goes to an error state if the `push` did not panic.

Panic behaviour. SEABMC provides control on how the unit proof is to behave when a panic is encountered. One mode treats a panic as an error (PANIC-ERROR). Another mode aborts execution on panic without error (PANIC-ABORT). For SEABMC, these modes are configured separately from a unit proof by configuring the Rust compiler with different panic handlers. Thus, the same unit proof can be run in different modes without modifying the proof code itself.

For example, in Fig. 2, we can remove line 9. Now when the proof is executed in PANIC-ABORT mode (with sufficiently large bound), some program executions will encounter a panic in `push`. This is because TINYVEC cannot grow beyond capacity. These executions will silently abort. The resulting verification conditions of the program may be easier to solve since panics are considered unreachable. Tool performance in PANIC-ABORT mode is discussed more in Section IV. Note that PANIC-ABORT mode also allows a proof design pattern to expect that a specific computation always panics. This is done in the form of an unreachable `verifier::error!()` as in line 24. Capturing that a specifc panic should occur in KANI is not possible as it reports *any* panic in the unit proof as an error. There is no PANIC-ABORT mode in KANI.

IV. BENCHMARKING

To examine how SEABMC works for practical Rust programs, we wrote unit proofs for four Rust libraries – TINYVEC, SMALLVEC, SEAVEC, and, RESULT-TYPE. The unit proofs cover common operations on data structures. The TINYVEC library provides a vector like data structure backed by an array. It panics if elements are added to it beyond capacity. The SMALLVEC library uses the stack for small vector sizes. Beyond this small size, SMALLVEC uses the system heap to store vectors. Many SMALLVEC internal operations are marked unsafe. The SEAVEC library is a minimal vector library to

[2]The WINCH case study uses a superficially different setup, but this is a historical accident.

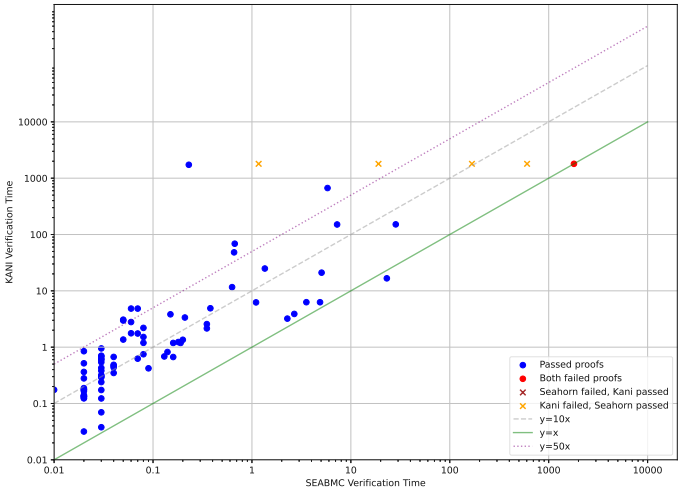

Fig. 3: Verification time (in sec.) using KANI vs SEABMC (using PANIC-ERROR).

serve as environment for verification of programs that require a vector implementation. In SEAVEC, memory is allocated on the heap and uses unsafe code. However, unlike SMALLVEC, there is no way to grow the vector beyond capacity. The RESULT-TYPE is a type used for returning and propagating errors in the Rust standard library. We have written 38 proofs for TINYVEC, 29 proofs for SMALLVEC, 9 for SEAVEC, and, 7 for RESULT-TYPE, for a total of 83 proofs. These proofs verify properties of common operations on the data structures. For example, for list data structures, the unit proof may check that a `push` increases the vector length by one and that push pushing to a list at capacity causes a panic. For RESULT-TYPE, the proof may check that a reference inside an `std::result` enum refers to the expected value. All proofs are checked for spatial memory safety by both SEABMC and KANI.

Setup. One of the goals of the benchmark data set is to be reproducible, debuggable, and extensible. For this we use the ReFrame [14] testing framework. ReFrame is mature, widely supported, and offers convenient test configuration syntax based on Python3. Some of its features we found useful were (1) the in-built design to separate test policy (what to test) from mechanism (how to test), (2) convenient syntax to search for strings in test output and operate on them – e.g., sum of all timings in `stdout`, and (3) staging for each test such that failures are reproducible for debugging

SEABMC and KANI. Figure 3 graphs verification time of KANI vs SEABMC as a scatter-plot on TINYVEC, SMALLVEC, SEAVEC, and, RESULT-TYPE. Both the axes are in log scale. The timeout for both KANI and SEABMC is set to 30 minutes. KANI times out for five unit proofs. SEABMC times out for one unit proof. We find that SEABMC is usually an order of magnitude faster than KANI.

We note that an apples-to-apples comparison of verification time between SEABMC and KANI is hard. We do not consider wall time because the compilation pipeline of the two tools differ. KANI plugs into the Rust compiler as a backend code generator. SEAHORN is invoked only after code generation to LLVM is complete. To compare time spent in bounded model

```
#[cfg_attr(kani, kani::proof)]                                  1
#[cfg_attr(kani, kani::unwind(3))]                              2
fn test_simplify_cfg() {                                        3
    let v: u8 = verifier::any!();                               4
    verifier::assume!(v < 3);                                   5
                                                                6
    for i in 0..v as usize {                                    7
      return;                                                   8
    }                                                           9
                                                                10
    let mut sentinel: u32 = verifier::any!();                   11
    // INV: v == 0 => sentinel >= 0                             12
    verifier::vassert!(sentinel >= (v as u32)*(v as u32));      13
}                                                               14
```

Fig. 4: LLVM optimization discharges all asserts in above program before SEABMC is invoked.

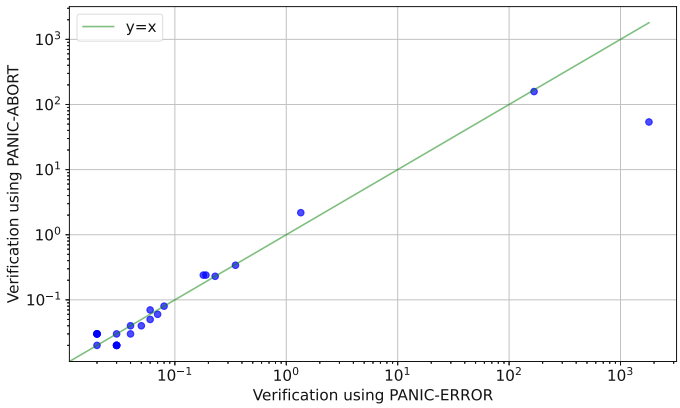

Fig. 5: Time (in sec.) to verify using SEABMC in PANIC-ABORT mode vs PANIC-ERROR mode.

checking, we consider verification time of KANI as logged by it after a series of post processing steps. The verification time of SEABMC is the tool running time after preprocessing.

While SEABMC shows consistently better performance than KANI on our benchmarks, there may be no single explaination for this. Both tools are multi-staged with unique strategies for generating verification conditions. Thus, it is difficult to *diff* where the added performance comes from. For SEABMC, we have found that the LLVM optimization passes that run as part of Rust compilation and during SEAHORN preprocessing can statically simplify the code considerably. We illustrate using the unit proof in Fig. 4. An execution may enter the loop in line 7 and exit early. Or, it may not enter the loop (`v = 0`). Thus, the only way to reach line 11 is when `v = 0`. Futher, the variable `sentinel` is an unsigned 32-bit number. With these two facts, the `vassert` in line 13 becomes trivially true since the type of `sentinel` ensures that it is always greater than equal to zero. This reasoning is done statically in the Rust + SEAHORN pipeline and the `vassert` is discharged before invoking VC generation. In our experiments, KANI invokes the solver indicating that it relies on the solver to reason as above. Invoking the solver is usually expensive, therefore, any pre-solver reasoning should improve verification time.

PANIC-ABORT vs PANIC-ERROR. In order to scale SEABMC to verify more complex code, we benchmark 24 unit proofs in TINYVEC to run in PANIC-ABORT vs PANIC-ERROR mode. The results are shown in Fig. 5 . On average, we see that PANIC-ABORT is 2.4x faster than PANIC-ERROR.

This is somewhat skewed by `test_retain` that times out with SEAHORN (30 minutes) in PANIC-ERROR mode and takes 54 seconds in PANIC-ABORT mode. As discussed in Section III, this speedup occurs because PANIC-ABORT leads to simpler unit proofs since weaker assumptions can be made.

V. APPLYING SEABMC TO WINCH

In this section, we present a case study in which we use SEAHORN (and SEABMC) to verify a real-world codebase: the WINCH compiler. WINCH compiles WebAssembly (WASM) code to native binaries. It is part of Wasmtime [15], an open-source project maintained by the Bytecode Alliance. We undertake its verification to support high-reliability compilation of WASM smart contracts [3] in the Stellar network [16], a popular blockchain.

Compared to the benchmarks of Section IV, we face several new challenges. First, the codebase is too large and complex to create unit proofs that exercise the end-to-end compilation of even small WASM programs. We, therefore, need to carve out a manageable SUT. This involves writing stubs for internal WINCH components and creating mixed concrete/symbolic contexts to exercise the SUT. Performing this surgery requires a good understanding of the internal structure of WINCH. Second, components that form the environment of the SUT likely maintain state invariants that the SUT depends on; if the context we create for the SUT does not satisfy those invariants, we get false-positive verification failures. Third, the WINCH codebase uses data-structures such as vectors with dynamic resizing, hash-maps, and hash-sets that are difficult for SEAHORN to execute symbolically. To make analysis feasible, we must replace these data-structures by stubs that are more tractable while exhibiting sufficiently rich behavior. Finally, to deliver long-term value, we need to be able to maintain the SUT and keep it up to date with changes to the upstream codebase.

In the next two sections, we describe these challenges in more detail, offer guidelines born from experience, and show that SEAHORN can realistically be applied to real-world code bases. Before we jump in, let us say a few more words about our motivations.

The Stellar network implements a fault-tolerant distributed ledger using state-machine replication[16]. Hundreds of replicas across the world each maintain a copy of the full ledger state, and users can access the ledger by invoking pieces of WASM code called smart contracts. Smart contracts are themselves user-provided, and users can register new smart contracts at any time. To keep their state synchronized, the replicas use a consensus protocol to repeatedly agree on the next operation to execute, e.g. adding a new smart contract to the system, or invoking an existing smart contract.

Currently, the replicas execute WASM code using the Wasmi interpreter [17]. However, in a proposed upgrade, replicas instead use WINCH to compile WASM code to native binaries that are then cached for later execution. Since users rely on the Stellar network for potentially high-value financial transactions, it is crucial that WINCH not crash, panic, or miscompile WASM code. Moreover, the state-machine replication protocol used in the Stellar network relies on identical, deterministic replica behavior. It is thus also important to verify that the behavior of WINCH does not depend on environment parameters (i.e. instruction set, operation system) that may vary across replicas.

For this case-study, we focus on verifying panic-freedom of WINCH's `x86_64` target. We therefore employ the PANIC-ERROR approach to panics, as described in Section III.

A. Carving out the System under Test

WINCH is a single-pass compiler that compiles WASM to native machine code. Fig. 6 illustrates at a high level the architecture of WINCH. The parser parses the input WASM code and the validator checks that it is valid, as defined by the WASM specification [18]. The result is an abstract syntax tree (AST) that has been validated. WINCH then traverses this AST in a single pass using the visitor design pattern [19]. For each AST node, a visit function corresponding to the node type starts a processing sequence involving the CODEGEN CONTEXT (responsible for high-level orchestration), MACRO ASSEMBLER (responsible for architecture-specific concerns) and ASSEMBLER (responsible for binary encodings), culminating with instructions being emitted to the MACHINE BUFFER.

WINCH borrows the parser and validator components from wasmparser [4] [20], and the MACHINE BUFFER component from the Cranelift compiler backend (also developed by the Wasmtime project). We focus our verification efforts on the WINCH-specific components exclusively. These components have not been battle-tested in other projects and may therefore harbor more issues. The SUT thus consists of the VISITOR's visit functions and the CODEGEN CONTEXT, MACRO ASSEMBLER, and ASSEMBLER components.

To prepare the SUT, we first modify WINCH to compile with Rust 1.64 in order to obtain LLVM 14 bitcode, the latest version currently supported by SEAHORN. To satisfy this requirement, we prune or downgrade several dependencies. Second, save for a few necessary type definitions, we remove the parser and the validator, and stub the MACHINE BUFFER using a minimal implementation simulating expected behavior. This allows us to isolate WINCH-specific components and obtain a tractable yet non-trivial SUT (about 3,300 lines).

Finally, we create a proof environment that sets up the necessary context for calling the SUT. Our entry points are the VISITOR's visit functions, which we provide with a mix of symbolic and concrete inputs. We also initialize the state of necessary WINCH components (e.g. the register allocator, stack frame, etc.) and set up various data structures. In all cases, we may use both concrete and symbolic data. To avoid spurious panics, we must constrain initial state and inputs to

[3] A program that executes predefined actions automatically when specific conditions are met.

[4] An open-source project maintained by the Bytecode Alliance.

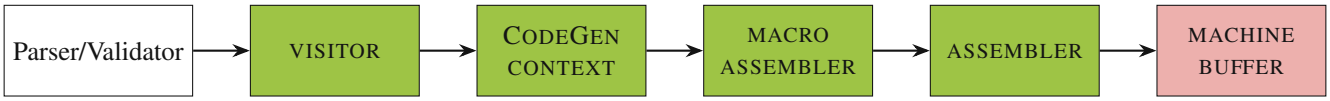

Fig. 6: WINCH Architecture showing, in order, trusted, verified and stubbed components.

```
#[no_mangle]                                                                    1
fn visit_cmp_ops() {                                                            2
  let vmoffsets = VMOffsets::new();                                             3
  let codegen_context = proof_core::setup_context(&vmoffsets);                  4
  let mut emission_context = codegen_context.for_emission();                    5
  let mut masm = proof_core::setup_masm();                                      6
  // SUT                                                                        7
  // invariant: second value on stack should be dst reg                         8
  let dst = Reg(PReg::new(2, regalloc2::RegClass::Int));                        9
  emission_context.stack.push(Val::Reg(TypedReg::i32(dst)));                    10
  let kind = IntCmpKind::from(nondet_u8());                                     11
  // call functions by operation width                                          12
  let v2 = nondet_u8();                                                         13
  let res = match v2 {                                                          14
    0 => {                                                                      15
      // invariant: need immediate/register at stack top                        16
      let val = nondet_i32();                                                   17
      emission_context.stack.push(Val::I32(val));                               18
      emission_context.i32_binop(&mut masm, |masm, dst, src, size| {            19
        masm.cmp_with_set(writable!(dst), src, kind, size)?;                    20
        Ok(TypedReg::i32(dst))                                                  21
      })                                                                        22
    },                                                                          23
    _ => {                                                                      24
      // invariant: need immediate/register at stack top                        25
      let val = nondet_i64();                                                   26
      emission_context.stack.push(Val::I64(val));                               27
      emission_context.i64_binop(&mut masm, |masm, dst, src, size| {            28
        masm.cmp_with_set(writable!(dst), src, kind, size)?;                    29
        Ok(TypedReg::i32(dst)) // Return value for cmp is an `i32`.             30
      })                                                                        31
    },                                                                          32
    };                                                                          33
    // check results                                                            34
    assert(res.is_ok());                                                        35
    emission_context.stack.peek().expect("value at stack top");                 36
    emission_context.stack.pop().unwrap().is_i32_const();                       37
    assert(emission_context.stack.inner().is_empty());                          38
}                                                                               39
```

Fig. 7: Unit proof for all comparison operations in the MACRO ASSEMBLER.

satisfy invariants that the SUT expects from upstream and external components. When such invariants are not documented, we must discover them during the verification process.

We now present a concrete example of unit proof for the VISITOR functions that handle comparison operations. At a high level, a comparison operation will pop two values from a stack, perform the comparison and push a result to the stack. We would like to prove that all visitor functions for comparison operations are panic free. Therefore, we need to setup the proof environment, choose a comparison kind, pick an operation width and check the returned results. This is the proof we show in Fig. 7.

Setting up the proof environment involves creating a CODEGEN CONTEXT and a MACRO ASSEMBLER (lines 3–6). CODEGEN CONTEXT is parametrized by two phases: PROLOGUE and EMISSION. These are responsible for setting up the function environment (frame) and emitting machine code respectively. We get a CODEGEN CONTEXT in the PROLOGUE phase by invoking `setup_context` using the default configuration for Virtual Machine (VM) offsets [5]. Since we are interested in end-to-end compilation of comparison operations, we change the context to be in the EMISSION phase. Finally, we create a MACRO ASSEMBLER using `setup_masm`. Once this environment has been setup, we can start proving the panic freedom of the system under test—a process that teaches us what invariants are preserved by WINCH. For example, the top value on the stack needs to represent a constant or a register. Moreover, the second value on the stack needs to be the destination register. We satisfy these invariants with lines 17–18, 26–27 and lines 9–10 respectively.

We have encapsulated the `setup_context` and `setup_masm` functions in a separate module for reusability across unit proofs. Their definitions are expanded in Fig. 8. The `setup_context` function takes the VM offsets and produces a CODEGEN CONTEXT in the PROLOGUE phase. It has the responsibility of creating a stack, a frame and a register allocator. We use a function signature with no parameters or returns, but any function signature can be used for the setup routine. The `setup_masm` function creates a MACRO ASSEMBLER for a specific ISA and shared flags specified in Wasmtime. This function has an important result that we discuss in Section V-B.

With the creation of a proof environment that satisfies the WINCH invariants, we continue with the unit proof in Fig. 7. The next step is to choose a comparison kind as shown in line 11. We use the `nondet_u8` function to generate a symbolic 8-bit unsigned integer [6]. By using the generated value, we are effectively choosing a comparison kind non-deterministically. Notice that the same approach is used when selecting the operation width to use (See line 13).

After the comparison functions have been called, we add assertions as shown in lines 35–38. For example, we assert that the result does not report any errors. We also assert that the stack has only one item which is a 32-bit constant. SEABMC returns *unsat* for this proof, giving us confidence that the 16 VISITOR functions that our proof covers are panic free and handle the stack correctly. In this example, we explore all reachable states because there are no unbounded loops to unroll.

B. Discovering Invariants with Executable Counter Examples

To better explain what we mean by executable counterexamples, we use our proof for the panic freedom of the MACRO ASSEMBLER's constructor. It is shown in the `setup_masm` function of Fig. 8. More specifically, we will explain why the proof needs a concrete value for the pointer size `ptr_size` that is passed into the constructor (See line 35).

[5]VM offsets are pointers to the host environment that a WASM program needs

[6]For details, see Section V-B

```rust
#[no_mangle]
pub fn setup_context<'a>(vmoffsets: &'a VMOffsets) ->
      CodeGenContext<'a, Prologue> {
  // stack setup
  let stack = Stack::new();
  // standard frame setup with 0 parameters/returns
  let sig = WasmFuncType::new(
    NoResizableVec::<WasmValType>::new(0),
    NoResizableVec::<WasmValType>::new(0)
  );
  let abi_sig = wasm_sig::<X64ABI>(&sig, 0, 0);
  let locals = DefinedLocals::new::<X64ABI>();
  let frame = Frame::new::<X64ABI>(&abi_sig, &locals.unwrap()).unwrap();
  // setup register allocator
  let gpr = RegBitSet::int(
    ALL_GPR.into(),
    NON_ALLOCATABLE_GPR.into(),
    usize::try_from(MAX_GPR).unwrap(),
  );
  let fpr = RegBitSet::float(
    ALL_FPR.into(),
    NON_ALLOCATABLE_FPR.into(),
    usize::try_from(MAX_FPR).unwrap(),
  );
  let regalloc = regalloc::RegAlloc::from(gpr, fpr);
  // CodeGen in Prologue phase
  let mut ctx = CodeGenContext::new(regalloc, stack, frame, &vmoffsets);
  return ctx;
}

#[no_mangle]
pub fn setup_masm() -> MacroAssembler {
  let isa_flags = cranelift_codegen::x64_settings::Flags::new();
  let shared_flags = cranelift_codegen::settings::Flags::new();
  // invariant: ptr_size has to be equal to 8
  let ptr_size = 8;
  let masm_64 = MacroAssembler::new(ptr_size, shared_flags, isa_flags);
  return masm_64.unwrap();
}
```

Fig. 8: Proof Environment Setup: `setup_context` and `setup_masm`.

We had initially defined `ptr_size` with the result of a function `nondet_u8` that takes no arguments and returns a symbolic value. This function is simply a wrapper around a function that SEAHORN understands for symbolic value generation, i.e. `__VERIFIER_nondet_u8`. Under this setup, SEABMC returns SAT, indicating that it finds a path in `setup_masm` that leads to a panic.

The counterexample (cex) pipeline in SEAHORN automatically extracts the values needed to reach the panic and generates the LLVM-IR file in Fig. 9. Lines 2–4 show that SEAHORN expects to provide two symbolic values. It maintains a global value that functions as an index into the array that contains the symbolic values. Using this setup, `__VERIFIER_nondet_u8` will read the index value into `%0`, increment and store it for future use, and return the symbolic value at index `%0`. This way, it returns the first symbolic value the first time it is called. And the second symbolic value the second time it is called.

Our scripts link this generated harness with the original input file to SEABMC and the SEAHORN runtime library to produce a standalone executable. This is the artifact that we call an executable counterexample. It provides a concrete, reproducible witness to the failure and can be run under standard debuggers, bridging the gap between symbolic verification and actionable diagnostics.

With this executable counterexample, the precise sequence of calls required to reach the panic are readily available. Stepping through the counterexample in `rust-lldb` leads us to a panic that encodes an important requirement: the compiler

```
// lines inlined or removed for simplicity
@0 = private constant [2 x i8] c"\00\F7"
@1 = private global i32 0
@__seahorn_cex_count = constant i32 2
define i8 @__VERIFIER_nondet_u8() {
entry:
  %0 = load i32, i32* @1, align 4
  %1 = add i32 %0, 1
  store i32 %1, i32* @1, align 4
  %2 = getelementptr inbounds [2 x i8], [2 x i8]* @0, i32 0, i32 %0
  %3 = load i8, i8* %2, align 1
  ret i8 %3
}
```

Fig. 9: Generated counter-example harness for `nondet_u8`.

expects to run on a 64-bit architecture. We document the invariant and use a concrete value to continue the proof.

C. Results and Takeaways

We have written 14 unit proofs, over three person months, meaningfully verifying 3307 LOC. Our proofs are split into four categories: `visit_setup`, `visit_cmp_ops`, `visit_arith` and `visit_funcs`. The categories are ordered in terms of difficulty. `visit_setup` covers parts of the ASSEMBLER and MACRO ASSEMBLER modules. `visit_cmp_ops` and `visit_arith` cover more parts of the ASSEMBLER and MACRO ASSEMBLER, in addition to parts of the CODEGEN CONTEXT module. Finally, `visit_funcs` covers parts of the CODEGEN CONTEXT that deal with complex data structures and their downstream components.

We show the results of our verification effort in Table I. For each proof category, we show the number of proofs in the collection, effort in man weeks, wall time and verification time in seconds. The `visit_setup` collection has two proofs that take less than one second to run. It took a person week to get the ASSEMBLER and MACRO ASSEMBLER to compile to LLVM 14 and get interesting verification results from SEABMC. The `visit_arith` and `visit_cmp_ops` proof collections have 10 proofs that cover 25 arithmetic and comparison VISITOR functions. These proof collections take less than 15s and 69s respectively. It took two person weeks to prototype verification strategies and leverage our learning to cover the VISITOR functions. The main challenge at this stage was maintaining a stack and a calling context for each proof. The `visit_funcs` collection has two proofs that cover the function signature and call sequence generation for locally defined functions. It took three person weeks to build the right proof environment and ensure proper handling of complex data structures. Despite the high human effort, the proofs take less than 6s to run.

It is important to note the difference in wall time and verification time in Table I. The wall time column represents the time it takes SEAHORN to process LLVM-IR, generate SEA-IR and run SEABMC. The verification time column is a subset of wall time that does not include input preprocessing. Note that most of the time reported for our proofs is not in the verification time column. Instead, a majority of the analysis time is spent simplify the verification problem by running preprocessing, optimization, and static analysis passes on LLVM bitcode. For example, in the case of `visit_arith`, there are 9 proofs that are checked by SEABMC. Out of the total time on these proofs, a fifth is spent on the first

	Proofs	Effort (person-weeks)	Wall Time (s)	Verification Time (s)
`visit_setup`	2	1	0.87	0.01
`visit_cmp_ops`	1	2	14.61	1.69
`visit_arith`	9	2	68.41	11.80
`visit_funcs`	2	3	5.97	0.27

TABLE I: Verification time for Winch proofs.

preprocessing step with LLVM passes. This illustrates how useful leveraging LLVM passes is for the simplification of the verification conditions generated by SEABMC. We believe this is a worthwhile tradeoff since it is likely to be an important contributor to SEAHORN's 10x advantage over Kani, as identified in Section IV.

It is noteworthy that the effort required for each proof collection increases linearly. This is a deliberate approach to make progress incrementally and manage the risk of investing time in untractable proof goals. Concretely, we start at the leafs of a visitor function's call tree and write proofs to establish the callee properties. Doing so iteratively for each caller/callee in the call tree maintains verification momentum without sacrificing the final outcome. This allows us to have more confidence about the panic freedom of WINCH, meaningfully verify more LOC and write more complex proofs after each time unit invested. We recommend this approach since it helped us build our understanding of the codebase incrementally while getting meaningful verification results.

The `visit_funcs` collection of proofs was the most time consuming for two main reasons. First, we had to find intricate invariants that need to be established by upstream components. The executable counterexamples generated by SEAHORN were instrumental to this effort. We found that WINCH requires parameters in a function signature to be stored in specific registers and stack offsets. For example, in our proof for functions with 8 parameters, 6 are put into specific registers and 2 are spilled to the stack, as required by the default WINCH calling conventions. We also learnt that the WINCH ABI uses special locals [7] that are spilled and treated as parameters. This means that we must reserve enough space for the parameters before invoking the SUT. Second, it is hard for SEAHORN to symbolically reason about HashMaps and HashSets. Therefore, we needed to place stubs for functionality that depended on HashMaps and HashSets.

VI. FUTURE DIRECTION

Our previous study [21] has shown that ownership in LLVM like representation can be used to generate efficient VC for BMC. SEABMC currently does not utilize ownership for VCGEN because all ownership information is lost when compiling Rust to LLVM. This is expected since LLVM does not have ownership concepts. We are developing a pipeline, called SEAURCHIN that extends LLVM with ownership semantics, and compiles Rust to this extended LLVM. With this effort we will retain the advantages of verifying a low-level, close to executable program representation. Simulataneously verification efficiency will improve by utilizing ownership semantics present in Rust programs.

For benchmarking, the number of unit proofs in VERIFY-RUST can be extended beyond the present number. In addition, VERIFIER-LIB can be extended to support other quality assurance methods (e.g., symbolic execution and fuzzing) to work with the same unit proof. This will be similar in spirit to [1] where multiple tools were benchmarked on the same unit proof.

To verify WINCH, we focused our efforts on a snapshot of the compiler. This results in a separate code-base that needs to be manually kept in sync with the main WINCH repository. Developing a user interface that makes it easy to integrate and maintain unit proofs in the main code-base is an interesting future challenge. This would be a tremendous contribution towards making the use of formal verification tools part of standard engineering practice.

VII. RELATED WORK

Verus [22], Prusti [23], Creusot [24], and Aenas [25] are deductive verifiers for Rust. These deductive tools can model complicated features of the language, like polymorphism, directly. This paper focuses on automatic verification of low-level memory manipulating programs.

The tool closest to SEABMC is KANI. KANI operates on Rust MIR programs, a higher level representation than LLVM IR. For KANI, there is an ever-present opportunity to leverage Rust high level information (e.g., ownership) in generating VC. For SEABMC to benefit similarly, LLVM IR has to be extended with semantics that capture high level information.

The SV-COMP [26] benchmark only considers tools that verify C programs. The benchmarks used in [9] are micro-benchmarks. There are no standardized benchmarks for Rust that we know of. The case studies in our work are a step towards filling this gap.

VIII. CONCLUSION

We show that SEABMC is fast, scalable and practical for bounded model checking of Rust code through two case studies. The first case study contributes 83 unit proofs for SMALLVEC, TINYVEC, SEAVEC, and RESULT-TYPE from the Rust standard library. On these benchmarks, SEABMC is faster than KANI by a factor of 10 or more. It is also more expressive in handling panic behaviour. The second case study contributes a methodology for verifying panic freedom in core parts of WINCH, an industrial WASM-to-executable compiler for various architectures. SEABMC is used to automatically check a collection of 14 proofs that meaningfully verify roughly $3,300$ LOC. Together, these case studies demonstrate our claims about SEABMC. SEAHORN is publicly available at https://github.com/seahorn/seahorn. The VERIFY-RUST case study is at https://github.com/seahorn/verify-rust. The WINCH case study is at https://github.com/jetafese/winch.

[7] VM Context pointers to the callee and caller

IX. Acknowledgments

We thank Adrian Jendo, Boris Jancic, Thomas Hart, and Liang Li for their contributions to the VERIFY-RUST benchmarks as part of their undergraduate research assistance-ships at the University of Waterloo.

References

[1] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel, "Verifying verified code," *Innov. Syst. Softw. Eng.*, vol. 18, no. 3, pp. 335–346, 2022. [Online]. Available: https://doi.org/10.1007/s11334-022-00443-9

[2] K. Developers. (2023) The kani book. [Online]. Available: https://model-checking.github.io/kani/

[3] D. Kroening, P. Schrammel, and M. Tautschnig, "CBMC: the C bounded model checker," *CoRR*, vol. abs/2302.02384, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.02384

[4] "Kani Rust Verifier Blog — model-checking.github.io," https://model-checking.github.io/kani-verifier-blog/, [Accessed 06-05-2025].

[5] J. Tafese, I. Garcia-Contreras, and A. Gurfinkel, "Btor2MLIR: A Format and Toolchain for Hardware Verification," in *Formal Methods in Computer Aided Design, FMCAD 2023*, 2023, p. 332.

[6] J. Tafese and A. Gurfinkel, "Efficient simulation for hardware model checking," in *Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, N. Bjørner, M. Heule, and A. Voronkov, Eds., vol. 100. EasyChair, 2024, pp. 136–146. [Online]. Available: /publications/paper/FDRF

[7] S. Priya, Y. Su, Y. Bao, X. Zhou, Y. Vizel, and A. Gurfinkel, "Bounded model checking for LLVM," in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 214–224. [Online]. Available: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_28

[8] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[9] M. S. Baranowski, S. He, and Z. Rakamaric, "Verifying rust programs with SMACK," in *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 11138. Springer, 2018, pp. 528–535. [Online]. Available: https://doi.org/10.1007/978-3-030-01090-4_32

[10] "GitHub - project-oak/rust-verification-tools: RVT is a collection of tools/libraries to support both static and dynamic verification of Rust programs. — github.com," https://github.com/project-oak/rust-verification-tools, [Accessed 08-05-2025].

[11] "FatLTO; LLVM documentation — llvm.org," http://llvm.org/docs/FatLTO.html, [Accessed 02-05-2025].

[12] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow at amazon web services," *Softw. Pract. Exp.*, vol. 51, no. 4, pp. 772–797, 2021. [Online]. Available: https://doi.org/10.1002/spe.2949

[13] "Symbol Mangling - The rustc book — doc.rust-lang.org," https://doc.rust-lang.org/rustc/symbol-mangling/index.html, [Accessed 01-05-2025].

[14] V. Karakasis, T. Manitaras, V. H. Rusu, R. Sarmiento-Pérez, C. Bignamini, M. Kraushaar, A. Jocksch, S. Omlin, G. Peretti-Pezzi, J. P. S. C. Augusto, B. Friesen, Y. He, L. Gerhardt, B. Cook, Z.-Q. You, S. Khuvis, and K. Tomko, "Enabling continuous testing of HPC systems using ReFrame," in *Tools and Techniques for High Performance Computing*, ser. Communications in Computer and Information Science, G. Juckeland and S. Chandrasekaran, Eds., vol. 1190. Cham, Switzerland: Springer International Publishing, Mar. 2020, pp. 49–68.

[15] Bytecode Alliance, "Wasmtime," `https://github.com/bytecodealliance/wasmtime/tree/50307096`.

[16] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, "Fast and secure global payments with Stellar," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, Oct. 2019, pp. 80–96.

[17] wasmi-labs, "Wasmi," `https://github.com/wasmi-labs/wasmi/tree/v0.38.0`.

[18] "WebAssembly Core Specification." [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," in *ECOOP' 93 — Object-Oriented Programming*, G. Goos, J. Hartmanis, and O. M. Nierstrasz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, vol. 707, pp. 406–431.

[20] Bytecode Alliance, "wasmparser," `https://crates.io/crates/wasmparser`.

[21] S. Priya and A. Gurfinkel, "Ownership in low-level intermediate representation," in *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, N. Narodytska and P. Rümmer, Eds. IEEE, 2024, pp. 292–300. [Online]. Available: https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_35

[22] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, "Verus: Verifying rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 286–315, 2023. [Online]. Available: https://doi.org/10.1145/3586037

[23] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, J. V. Deshmukh, K. Havelund, and I. Perez, Eds., vol. 13260. Springer, 2022, pp. 88–108. [Online]. Available: https://doi.org/10.1007/978-3-031-06773-0_5

[24] X. Denis, J. Jourdan, and C. Marché, "Creusot: A foundry for the deductive verification of rust programs," in *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings*, ser. Lecture Notes in Computer Science, A. Riesco and M. Zhang, Eds., vol. 13478. Springer, 2022, pp. 90–105. [Online]. Available: https://doi.org/10.1007/978-3-031-17244-1_6

[25] "Aeneas, a Verification Framework for Rust — aeneasverif.github.io," https://aeneasverif.github.io/, [Accessed 08-05-2025].

[26] D. Beyer, "State of the art in software verification and witness validation: SV-COMP 2024," in *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, ser. Lecture Notes in Computer Science, B. Finkbeiner and L. Kovács, Eds., vol. 14572. Springer, 2024, pp. 299–329. [Online]. Available: https://doi.org/10.1007/978-3-031-57256-2_15